实验报告

目录

实验目的

1. 实验任务 2.1：正规表达式转 NFA 算法及实现
    1. 掌握正规表达式与有限自动机的基本概念和转换方法。
    2. 了解非确定有限自动机（NFA）的构建过程。
    3. 熟悉编程实现正规表达式到 NFA 转换的算法。
    4. 提高编程能力和算法设计的技能。
    5. 增加可视化功能，利用 Graphviz 等库，实现 NFA 和 DFA 的可视化展示。
2. 实验任务 2.2 NFA 转 DFA 算法及实现
    1. 掌握非确定有限自动机（NFA）与确定有限自动机（DFA）的基本概念及其转换方法。
    2. 了解 NFA 到 DFA 转换过程中的子集构造算法。
    3. 实现 NFA 到 DFA 的转换算法，并验证 DFA 的正确性。
    4. 设计合理的数据结构，延续上一次实验的结构，以便为后续 DFA 最小化实验任务做好准备。
    5. 提高编程能力及算法设计和优化的技能
3. 实验任务 2.3 DFA 最小化算法及实现
    1. 掌握确定有限自动机（DFA）的最小化原理和算法，尤其是 Hopcroft 算法（即课上所讲的“求异法”）。
    2. 学习 DFA 状态等价性的判定方法，理解最小化过程中的分割和合并策略。
    3. 实现 DFA 最小化算法，并验证最小化 DFA 的正确性。
    4. 延续前两次实验的设计，确保数据结构能贯通整个自动机系列实验。
    5. 提高算法优化和编程实现能力，增强对编译原理的理解

实验内容

1. 实验任务 2.1：正规表达式转 NFA 算法及实现
    - 解析输入的正规表达式。
    - 构建对应的 NFA，包括处理基本符号、连接、并联（或操作）、闭包（星号操作）等运算。
    - 设计并实现合理的数据结构表示 NFA，如状态集合、转移关系、初始状态和接受状态。
    - 对 NFA 进行模拟，验证其是否接受给定的输入字符串。
2. 实验任务 2.2 NFA 转 DFA 算法及实现
    - 理解子集构造算法的原理，包括 ε-闭包的计算和状态集合的映射。
    - 利用子集构造算法，将 NFA 转换为 DFA。
    - 设计并实现 DFA 的数据结构，确保其能够表示状态集合、状态转换、初始状态和接受状态。
    - 验证 DFA 的正确性，对比 DFA 与 NFA 在同一组测试输入上的匹配结果。
3. 实验任务 2.3 DFA 最小化算法及实现
    - 理解 Hopcroft 算法的基本原理，包括状态等价的判定标准和状态合并的方法。
    - 实现 Hopcroft 算法，将原 DFA 简化为等价的最小化 DFA。
    - 设计合理的数据结构表示最小化后的 DFA，确保其与前两次实验的 NFA 和 DFA 数据结构保持一致。
    - 验证最小化 DFA 的正确性，确保其接受的语言与原 DFA 相同。

算法实现和数据结构设计

正规表达式转 NFA

类 `State`

```
class State:
  id_counter = 0
  def __init__(self, name=None):
    self.name = name or f"S{State.id_counter}"
    State.id_counter += 1
    self.transitions: dict[str, list[State]] = {}

  def add_transition(self, input_char, state):
    self.transitions.setdefault(input_char, []).append(state)

  def __str__(self):
    return self.name
```

用途

类 `State` 用于表示 NFA 或 DFA 中的一个状态。每个状态具有一个独特的名称和转移关系的字典，该字典存储从当前状态出发到达其他状态的路径。

属性

- `name`：状态的名称，如未提供则自动生成，如 "S0", "S1" 等。
- `transitions`：一个字典，键为输入字符（字符串类型），值为目标状态的列表。这表示从当前状态通过相应的输入字符可以到达的一个或多个状态。

方法

- `__init__(self, name=None)`：构造函数，用于初始化状态对象。如果未指定 `name`，则自动生成一个唯一的状态名。
- `add_transition(self, input_char, state)`：添加一个转移关系到 `transitions` 字典。如果给定的输入字符 `input_char` 已存在于字典中，则将状态添加到对应列表中；否则，创建一个新列表。
- `__str__(self)`：返回状态的字符串表示，便于在打印和调试中使用。

类 `NFA`

```
import graphviz

class NFA:
  def __init__(self, states: list[State] = []):
    self.states: list[State] = states
    self.start_state: State = states[0] if states else None
    self.accept_state: State = states[-1] if states else None

  def add_state(self, state: State):
    """添加状态到NFA"""
    self.states.append(state)
```

```
  def set_start_state(self, state: State):
    """设置起始状态"""
    self.start_state = state

  def set_accept_state(self, state: State):
    """设置接受状态"""
    self.accept_state = state

  def epsilon_closure(self, states: list[State]) -> set[State]:
    """计算给定状态集的epsilon闭包"""
    stack = list(states)
    closure = set(stack)
    while stack:
      state = stack.pop()
      if None in state.transitions:
        for next_state in state.transitions[None]:
          if next_state not in closure:
            closure.add(next_state)
            stack.append(next_state)
    return closure

  def move(self, states: list[State], input_char) -> set[State]:
    """ 返回从给定状态集出发，经过input_char可以到达的状态集 """
    next_states = set()
    for state in states:
      if input_char in state.transitions:
        next_states.update(state.transitions[input_char])
    return next_states

  def simulate(self, input_string: str):
    """模拟NFA处理输入字符串，返回是否接受该字符串"""
    current_states = self.epsilon_closure([self.start_state])
    for char in input_string:
      next_states = set()
      for state in current_states:
        if char in state.transitions:
          for next_state in state.transitions[char]:
            next_states.update(self.epsilon_closure([next_state]))
      current_states = next_states
    return self.accept_state in current_states

  def visualize(self, filename: str):
    """可视化NFA，使用Graphviz"""
    dot = graphviz.Digraph()
    dot.node("", shape="none")
    dot.edge("", str(self.start_state), label='start')
    states = [self.start_state]
    for state in self.states:
      shape = 'doublecircle' if state == self.accept_state else 'circle'
      dot.node(str(state), shape=shape)
      for char, states in state.transitions.items():
        label = char if char is not None else 'ε'
        for next_state in states:
          dot.edge(str(state), str(next_state), label=label)
    dot.render(filename, format="png", view=False)
```

```
  def copy(self):
    """创建并返回此NFA的深拷贝"""
    state_map = {state: State() for state in self.states}
    new_nfa = NFA([])
    for state in self.states:
      new_state = state_map[state]
      new_nfa.add_state(new_state)
      for char, states in state.transitions.items():
        for target_state in states:
          new_state.add_transition(char, state_map[target_state])
    if self.start_state:
      new_nfa.set_start_state(state_map[self.start_state])
    if self.accept_state:
      new_nfa.set_accept_state(state_map[self.accept_state])
    return new_nfa
```

用途

类 `NFA` 用于表示一个非确定有限自动机。它包含一组状态，以及特定的起始状态和接受状态。

属性

- `states`: 存储自动机中所有状态的列表。
- `start_state`: 自动机的起始状态。
- `accept_state`: 自动机的接受状态，即识别结束时可以处于的状态。

方法

- `__init__(self, states=[])`: 构造函数，初始化一个 NFA 实例，可以选择性地传入状态集合，将状态集合转换为 NFA。
- `add_state(self, state)`: 向 NFA 添加一个新的状态。
- `set_start_state(self, state)`: 设置 NFA 的起始状态。
- `set_accept_state(self, state)`: 设置 NFA 的接受状态。
- `epsilon_closure(self, states)`: 计算给定状态集合的 ε-闭包。此方法通过迭代添加所有由 ε-转移可达的状态来实现。
- `move(self, states, input_char)`: 根据给定的输入字符，返回从指定状态集出发可达的新状态集合。
- `simulate(self, input_string)`: 模拟 NFA 处理输入字符串。此方法通过从起始状态的 ε-闭包开始，并对每个字符更新当前状态集合，最后检查接受状态是否在最终状态集合中。
- `visualize(self, filename)`: 使用 Graphviz 库生成 NFA 的可视化图形。此方法为每个状态和转移关系创建图形节点和边。
- `copy(self)`: 创建并返回 NFA 的深拷贝。此方法对每个状态进行复制，并确保复制的 NFA 保持原有的转移关系和状态属性。

这两个类为正规表达式到 NFA 的转换和 NFA 到 DFA 的转换提供了基础结构。使用这些数据结构，可以有效地模拟和操作有限自动机，支持从简单到复杂的正规表达式的各种操作，为编译原理和模式匹配的实现提供强有力的工具。

算法

1. **添加显式连接操作符**：这一步是为了确保正规表达式中的每个字符和操作符之间的隐式连接关系（如`ab`变为`a.b`）都被显式地表示出来，方便后续转换为后缀表达式和构建 NFA。

```
class Regex:
  def __init__(self, pattern: str):
    def add_explicit_concat_operator(expression: str):
      """添加显示连接符号"""
      output = []
      unaryOperators = {"*", "+", "?"}
      binaryOperators = {"|"}
      operators = unaryOperators.union(binaryOperators)
      for i in range(len(expression) - 1):
        output.append(expression[i])
        assert not (
          expression[i] in unaryOperators
          and expression[i + 1] in unaryOperators
        )
        assert not (
          expression[i] in binaryOperators
          and expression[i + 1] in binaryOperators
        )
        assert not (
          expression[i] in binaryOperators
          and expression[i + 1] in unaryOperators
        )
        # 如果当前字符和下一个字符之间应该有连接操作符，则插入'.'。
        if (expression[i] not in binaryOperators and expression[i] != "(") and (
          expression[i + 1] not in operators and expression[i + 1] != ")"
        ):
          output.append(".")
      output.append(expression[-1])
      return "".join(output)

    self.pattern: str = add_explicit_concat_operator(pattern)
```

2. **中缀转后缀表达式**：将正规表达式由中缀形式转换为后缀形式，后缀形式有助于简化后续的自动机构建过程，因为它避免了处理运算符优先级和括号的复杂性。

```
def to_postfix(self):
  """将正则表达式转换为后缀表达式"""
  precedence = {".": 2, "|": 1}  # '.'表示连接操作
  output = []
  stack = []
  for char in self.pattern:
    if char in precedence:
      while (
        stack
        and stack[-1] != "("
        and precedence[stack[-1]] >= precedence[char]
      ):
        output.append(stack.pop())
      stack.append(char)
```

```
    elif char == "(":
      stack.append(char)
    elif char == ")":
      while stack and stack[-1] != "(":
        output.append(stack.pop())
      stack.pop()
    else:
      output.append(char)
  while stack:
    output.append(stack.pop())
  return "".join(output)
```

3. **构建 NFA**：基于后缀表达式，使用栈结构来逐步构建 NFA。每个操作符对应一种操作，将相关的 NFA 片段弹出栈，应用操作后将结果 NFA 压回栈中。

```
def create_basic_nfa(self, char):
  """实现单个符号的NFA"""
  start_state = State()
  accept_state = State()
  start_state.add_transition(char, accept_state)
  return NFA([start_state, accept_state])

def apply_closure(self, nfa: NFA):
  """实现闭包操作: NFA*"""
  start_state = State()
  accept_state = State()
  start_state.add_transition(None, nfa.start_state)
  start_state.add_transition(None, accept_state)
  nfa.accept_state.add_transition(None, nfa.start_state)
  nfa.accept_state.add_transition(None, accept_state)
  nfa.add_state(start_state)
  nfa.add_state(accept_state)
  nfa.set_start_state(start_state)
  nfa.set_accept_state(accept_state)
  return nfa

def apply_plus(self, nfa: NFA):
  """实现一次或多次重复: NFA+ 相当于 NFA . NFA*"""
  nfa_copy = nfa.copy()
  nfa_copy.visualize("test")
  return self.apply_concatenation(nfa_copy, self.apply_closure(nfa))

def apply_question(self, nfa: NFA):
  """实现零次或一次出现: NFA? 相当于 (ε|NFA)"""
  start_state = State()
  accept_state = State()
  start_state.add_transition(None, accept_state)
  epsilon_nfa = NFA([start_state, accept_state])
  return self.apply_union(epsilon_nfa, nfa)

def apply_union(self, nfa1: NFA, nfa2: NFA):
  """实现并联操作: NFA1 | NFA2"""
  start_state = State()
  accept_state = State()
  start_state.add_transition(None, nfa1.start_state)
```

```
    start_state.add_transition(None, nfa2.start_state)
    nfa1.accept_state.add_transition(None, accept_state)
    nfa2.accept_state.add_transition(None, accept_state)
    return NFA([start_state] + nfa1.states + nfa2.states + [accept_state])

def apply_concatenation(self, nfa1: NFA, nfa2: NFA):
    """实现连接操作: NFA1 . NFA2"""
    nfa1.accept_state.add_transition(None, nfa2.start_state)
    nfa = NFA(nfa1.states + nfa2.states)
    nfa.set_start_state(nfa1.start_state)
    nfa.set_accept_state(nfa2.accept_state)
    return nfa

def to_nfa(self):
    postfix = self.to_postfix()
    stack = []
    for char in postfix:
        if char == "*":
            # 应用闭包操作
            nfa = stack.pop()
            stack.append(self.apply_closure(nfa))
        elif char == "+":
            # 应用正闭包操作
            nfa = stack.pop()
            stack.append(self.apply_plus(nfa))
        elif char == "?":
            # 应用可选操作
            nfa = stack.pop()
            stack.append(self.apply_question(nfa))
        elif char == "|":
            # 应用并操作
            nfa2 = stack.pop()
            nfa1 = stack.pop()
            stack.append(self.apply_union(nfa1, nfa2))
        elif char == ".":
            # 应用连接操作
            nfa2 = stack.pop()
            nfa1 = stack.pop()
            stack.append(self.apply_concatenation(nfa1, nfa2))
        else:
            # 对于基本符号，创建一个基本NFA
            stack.append(self.create_basic_nfa(char))
    return stack.pop()
```

NFA 转 DFA（子集构造算法）

类 `DFA`

```
import graphviz

class DFA:
    def __init__(self, nfa: NFA=None):
        """初始化DFA。如果提供NFA，将其转换为DFA"""
        self.states: list[State] = []
        self.start_state: State = None
```

```
    self.accept_states: list[State] = []
    if nfa:
      self.initialize_from_nfa(nfa)

  def initialize_from_nfa(self, nfa: NFA):
    """根据给定的NFA构建DFA"""
    initial_closure = nfa.epsilon_closure([nfa.start_state])
    start_state = State()
    self.add_state(start_state, nfa.start_state == nfa.accept_state)
    self.set_start_state(start_state)

    unmarked = [(start_state, initial_closure)]
    marked = {}
    marked[frozenset(initial_closure)] = start_state

    while unmarked:
      current_dfa_state, current_nfa_states = unmarked.pop(0)
      for input_char in set(char for state in current_nfa_states for char in
state.transitions if char is not None):
        new_nfa_states = nfa.epsilon_closure(nfa.move(current_nfa_states,
input_char))
        state_set = frozenset(new_nfa_states)
        if state_set not in marked:
          new_dfa_state = State()
          self.add_state(new_dfa_state, any(s == nfa.accept_state for s in
new_nfa_states))
          marked[state_set] = new_dfa_state
          unmarked.append((new_dfa_state, new_nfa_states))
        current_dfa_state.add_transition(input_char, marked[state_set])

  def add_state(self, state: State, is_accept=False):
    """添加一个状态到DFA，并标记是否为接受状态"""
    self.states.append(state)
    if is_accept:
      self.accept_states.append(state)

  def set_start_state(self, state: State):
    """设置DFA的起始状态"""
    self.start_state = state

  def simulate(self, input_string: str):
    """模拟DFA，检查是否接受给定的输入字符串"""
    current_state = self.start_state
    for char in input_string:
      if char in current_state.transitions:
        current_state = current_state.transitions[char][0]
      else:
        return False
    return current_state in self.accept_states

  def visualize(self, filename:str):
    """使用Graphviz可视化DFA"""
    dot = graphviz.Digraph()
    dot.node("", shape="none")
    dot.edge("", str(self.start_state), label='start')
    for state in self.states:
```

```
      shape = 'doublecircle' if state in self.accept_states else 'circle'
      dot.node(str(state), shape=shape)
      for char, next_states in state.transitions.items():
        for next_state in next_states:
          dot.edge(str(state), str(next_state), label=str(char))
    dot.render(filename, format="png", view=False)


  def minimize(self):
    """使用Hopcroft算法求异法最小化DFA"""
    symbols = {symbol for state in self.states for symbol in state.transitions}

    accept = frozenset(self.accept_states)
    non_accept = frozenset(self.states) - accept
    P = {accept, non_accept}
    W = P.copy()

    while W:
      A = W.pop()
      for symbol in symbols:
        X = {state for state in self.states if any(next_state in A for next_state
in state.transitions.get(symbol, []))}
        if not X:
          continue

        for Y in P.copy():
          intersection = Y & X
          difference = Y - X
          if intersection and difference:
            P.remove(Y)
            P.add(frozenset(intersection))
            P.add(frozenset(difference))
            if Y in W:
              W.remove(Y)
              W.add(frozenset(intersection))
              W.add(frozenset(difference))
            else:
              if len(intersection) <= len(difference):
                W.add(frozenset(intersection))
              else:
                W.add(frozenset(difference))

    new_states = {frozenset(group): State() for group in P}
    new_start_state = next(new_states[group] for group in P if self.start_state
in group)
    new_accept_states = {new_states[group] for group in P if any(s in accept for
s in group)}

    for group, new_state in new_states.items():
      for state in group:
        for sym, destinations in state.transitions.items():
          dest_group = next(g for g in P if any(d in g for d in destinations))
          if new_states[dest_group] not in new_state.transitions.get(sym, []):
            new_state.add_transition(sym, new_states[dest_group])

    new_dfa = DFA()
```

```
    new_dfa.states = list(new_states.values())
    new_dfa.start_state = new_start_state
    new_dfa.accept_states = list(new_accept_states)
    return new_dfa
```

用途

类 `DFA` 用于表示确定有限自动机（DFA），这是一种每个状态对于每个可能的输入符号都有一个唯一确定的转移状态的自动机。与非确定有限自动机（NFA）相比，DFA 在处理输入字符串时更直接且效率更高，因为不需要处理多个可能的转移状态。

属性

- `states`：存储 DFA 中所有状态的列表。
- `start_state`：DFA 的起始状态。
- `accept_states`：一个包含所有接受状态的列表，接受状态是在识别结束时可以处于的状态。

方法

- `__init__(self, nfa=None)`：构造函数，可以选择性地通过一个 NFA 初始化 DFA，如果提供了 NFA，则立即启动从 NFA 到 DFA 的转换过程。
- `initialize_from_nfa(self, nfa)`：使用子集构造法将给定的 NFA 转换成 DFA。此过程涉及计算 NFA 状态的 ε-闭包并构建新的 DFA 状态，确保每个 DFA 状态对于每个输入符号有一个确定的转移。
- `add_state(self, state, is_accept=False)`：向 DFA 添加一个状态，并可指定该状态是否为接接受状态。
- `set_start_state(self, state)`：设置 DFA 的起始状态。
- `simulate(self, input_string)`：模拟 DFA 处理输入字符串，返回该字符串是否被接受。这是通过从起始状态开始，并根据输入字符串顺序迁移状态来完成的。
- `visualize(self, filename)`：使用 Graphviz 工具生成 DFA 的可视化图形。这对理解 DFA 的结构和调试非常有用。
- `minimize(self)`：使用 Hopcroft 算法对 DFA 进行最小化。该算法通过识别等价状态（不区分输入字符串处理结果的状态）并合并这些状态来简化 DFA。

这些详细的描述和操作说明为理解和实现 DFA 提供了全面的指导，确保了在正规表达式处理、模式匹配和其他自动机应用中的高效和准确性。

算法详细解释

1. 计算初始状态的 ε-闭包

此步骤的目的是确定 DFA 的起始状态，它是由 NFA 的起始状态的 ε-闭包构成的。ε-闭包是指从某状态出发，仅通过 ε-转移（即不消耗输入符号的转移）所能到达的所有状态的集合。

```
def epsilon_closure(self, states):
  stack = list(states)  # 将初始状态集合转换成栈，用于深度优先搜索
  closure = set(stack)  # 初始化闭包集合，起始包含所有初始状态
  while stack:
    state = stack.pop()  # 弹出栈顶元素进行探索
    if None in state.transitions:  # 检查当前状态是否有ϵ-转移
      for next_state in state.transitions[None]:  # 遍历通过ϵ-转移可达的所有状态
        if next_state not in closure:  # 如果状态未被访问过
          closure.add(next_state)  # 加入闭包集合
          stack.append(next_state)  # 将新状态加入栈，以便后续探索
  return closure  # 返回计算的ϵ-闭包
```

2. 迭代构建 DFA

在这一步中，我们使用子集构造算法（也称为幂集构造算法）将 NFA 转换为 DFA。这一过程涉及创建 DFA 的状态，每个状态对应 NFA 状态集合的一个子集。每个 DFA 状态表示 NFA 的一个或多个状态的组合。

```
class DFA:
  def __init__(self, nfa: NFA=None):
    """初始化DFA。如果提供NFA，将其转换为DFA"""
    self.states: list[State] = []
    self.start_state: State = None
    self.accept_states: list[State] = []
    if nfa:
      self.initialize_from_nfa(nfa)

def initialize_from_nfa(self, nfa):
  initial_closure = nfa.epsilon_closure([nfa.start_state])  # 计算 NFA 起始状态的ϵ-闭包
  start_state = State()  # 创建 DFA 的初始状态
  self.add_state(start_state, nfa.start_state == nfa.accept_state)  # 添加初始状态，判断是否为接受状态
  self.set_start_state(start_state)  # 设置 DFA 的起始状态

  unmarked = [(start_state, initial_closure)]  # 初始化未标记列表，存储新的 DFA 状态和对应的 NFA 状态集
  marked = {frozenset(initial_closure): start_state}  # 标记已处理的 NFA 状态集合和对应的 DFA 状态

  while unmarked:  # 当存在未处理的 DFA 状态时
    current_dfa_state, current_nfa_states = unmarked.pop(0)  # 取出一个未处理的状态及其对应的 NFA 状态集
    for input_char in set(char for state in current_nfa_states for char in state.transitions if char is not None):  # 遍历所有可能的输入字符
      new_nfa_states = nfa.epsilon_closure(nfa.move(current_nfa_states, input_char))  # 计算在该输入字符下可达的新 NFA 状态集的 ϵ-闭包
      state_set = frozenset(new_nfa_states)  # 创建新状态集合的不可变版本，用于记录和查询
      if state_set not in marked:  # 如果新状态集未被处理
        new_dfa_state = State()  # 创建新的 DFA 状态
        self.add_state(new_dfa_state, any(s == nfa.accept_state for s in new_nfa_states))  # 判断并标记为接受状态
        marked[state_set] = new_dfa_state  # 记录新状态
        unmarked.append((new_dfa_state, new_nfa_states))  # 添加到未处理列表
```

```
            current_dfa_state.add_transition(input_char, marked[state_set])  # 建立从当前 DFA 状态到新 DFA 状态的转移
```

这两个过程合起来，形成了从 NFA 到 DFA 的完整转换，确保了在处理输入字符串时，每个状态的转移都是确定的，这是 DFA 的主要特点。通过这种方式，DFA 能够更高效地处理字符串匹配问题。

DFA 最小化（Hopcroft 算法）

原理和步骤

Hopcroft 算法是 1971 年由 John Hopcroft 提出的，用于最小化确定性有限自动机（DFA）。该算法的核心思想是通过分割状态集合来逐步减少状态数量，从而找到最小化的 DFA。算法主要步骤如下：

1. 初始化：将状态集合分为接受状态和非接受状态两部分，构建初始状态分割。
2. 细化分割
    - 对于每个输入符号，检查哪些状态在该符号下转移到不同的状态。
    - 根据转移结果细化当前状态分割，合并那些能够在相同输入符号下转移到相同状态的状态。
3. 重复上述步骤，直到无法再细分为止。

算法详细解释

```python
def minimize(self):
  # 提取所有出现在转移中的符号
  symbols = {symbol for state in self.states for symbol in state.transitions}
  # 将状态分为接受状态和非接受状态两组
  accept = frozenset(self.accept_states)
  non_accept = frozenset(self.states) - accept
  # 初始化划分 P 和待处理集合 W
  P = {accept, non_accept}
  W = P.copy()

  # 当待处理集合非空时循环
  while W:
    A = W.pop()  # 取出一个待处理的组 A
    # 对于每个符号
    for symbol in symbols:
      # X 是在读入 symbol 后可以转移到 A 中某状态的所有状态的集合
      X = {state for state in self.states if any(next_state in A for next_state
in state.transitions.get(symbol, []))}
      if not X:
        continue
      # 对于当前划分中的每个集合 Y
      for Y in P.copy():
        intersection = Y & X  # 交集
        difference = Y - X  # 差集
        # 如果 Y 被 X 分割成两个非空集合
        if intersection and difference:
          P.remove(Y)  # 从 P 中移除当前的 Y
          P.add(frozenset(intersection))  # 添加交集和差集作为新的划分
          P.add(frozenset(difference))
          # 更新待处理集合 W
          if Y in W:
            W.remove(Y)
```

```
            W.add(frozenset(intersection))
            W.add(frozenset(difference))
          else:
            # 将较小的集合添加到 W 中，优化性能
            if len(intersection) <= len(difference):
              W.add(frozenset(intersection))
            else:
              W.add(frozenset(difference))

  # 创建新的状态集合，每个新状态代表一个等价类
  new_states = {frozenset(group): State() for group in P}
  # 确定新的起始状态
  new_start_state = next(new_states[group] for group in P if self.start_state in
group)
  # 确定新的接受状态集
  new_accept_states = {new_states[group] for group in P if any(s in accept for s
in group)}

  # 重建状态转移
  for group, new_state in new_states.items():
    for state in group:
      for sym, destinations in state.transitions.items():
        dest_group = next(g for g in P if any(d in g for d in destinations))
        new_state.add_transition(sym, new_states[dest_group])

  # 创建并返回新的 DFA 实例
  new_dfa = DFA()
  new_dfa.states = list(new_states.values())
  new_dfa.start_state = new_start_state
  new_dfa.accept_states = list(new_accept_states)
  return new_dfa
```

此代码通过逐步细分 DFA 的状态集合，并在每一步中尝试分割每个当前的等价类，以便不同的输入导致不同的输出状态集。这种细分过程持续进行，直到无法进一步分割为止。这样，算法最终得到的每个状态集合都是最小的，不能再被细分，从而达到了 DFA 最小化的目的。

测试用例与结果分析

```
def main():
  patterns = ["ab", "a|c", "a(b|c)", "(a|b)*", "(a|b)+", "ab+c?", "(ab)*|c+",
"b(a|b)*aa", "(a|b)*abb"]
  strings = ["ab", "abc", "abcc", "c", "abbbbb", "abab", "bababababaaa"]

  for pattern in patterns:
    regex = Regex(pattern)
    print(f"\nTesting pattern: {regex.pattern}")
    nfa = regex.to_nfa()
    dfa = DFA(nfa)
    mini_dfa = dfa.minimize()
    nfa.visualize('result/nfa_' + pattern)
    dfa.visualize('result/dfa_' + pattern)
    mini_dfa.visualize('result/dfa_minimize_' + pattern)
    for string in strings:
      nfa_result = nfa.simulate(string)
      dfa_result = dfa.simulate(string)
```

```
        mini_dfa_result = mini_dfa.simulate(string)
        assert(nfa_result == dfa_result)
        assert(mini_dfa_result == dfa_result)
        if (nfa_result):
          print(f"\033[32mMatched: {string}\033[0m")
        else:
          print(f"\033[31mUnmatched: {string}\033[0m")

if __name__ == "__main__":
  main()
```

本实验通过一系列正规表达式，测试了从正规表达式到 NFA，再从 NFA 到 DFA，以及 DFA 到最小化 DFA 的整个转换过程。以下是实验中使用的正规表达式和测试字符串：

正规表达式列表

- `"ab"`：匹配字符串 "ab"
- `"a|c"`：匹配 "a" 或 "c"
- `"a(b|c)"`：匹配 "ab" 或 "ac"
- `"(a|b)*"`：匹配由 "a" 或 "b" 组成的任意长度的字符串（包括空字符串）
- `"(a|b)+"`：匹配由 "a" 或 "b" 组成的至少一个字符的字符串
- `"ab+c?"`：匹配 "ab" 后跟一个或多个 "b"，可选一个 "c"
- `"(ab)*|c+"`：匹配 "ab" 的任意重复次数，或至少一个 "c"
- `"b(a|b)*aa"`：匹配以 "b" 开头，后接任意数量的 "a" 或 "b"，并以 "aa" 结尾
- `"(a|b)*abb"`：匹配任意数量的 "a" 或 "b"，以 "abb" 结尾

测试字符串列表

- `"ab"`
- `"abc"`
- `"abcc"`
- `"c"`
- `"abbbbb"`
- `"abab"`
- `"babababababaaa"`

测试结果分析

每个正规表达式都生成了对应的 NFA、DFA 和最小化 DFA，这些自动机被用来匹配测试字符串。以下是关键的结果验证点：

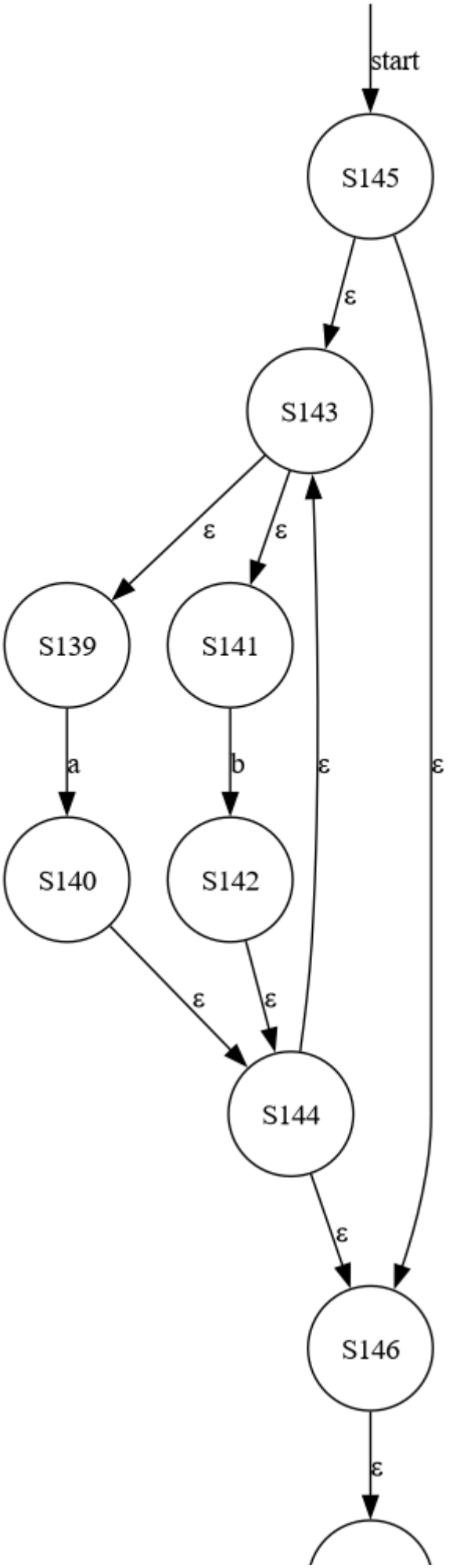
1. **NFA, DFA, 和最小化 DFA 的一致性**：对于每个正规表达式和每个测试字符串，验证 NFA, DFA, 和最小化 DFA 的模拟结果是否一致。这确保了转换的正确性。
2. **正确的字符串识别**：根据每个正规表达式的定义，验证自动机是否正确地接受或拒绝每个测试字符串。例如，正规表达式 `"ab"` 只应该接受字符串 `"ab"`，而正规表达式 `"(a|b)*"` 应该接受所有测试字符串中由 "a" 或 "b" 组成的部分。

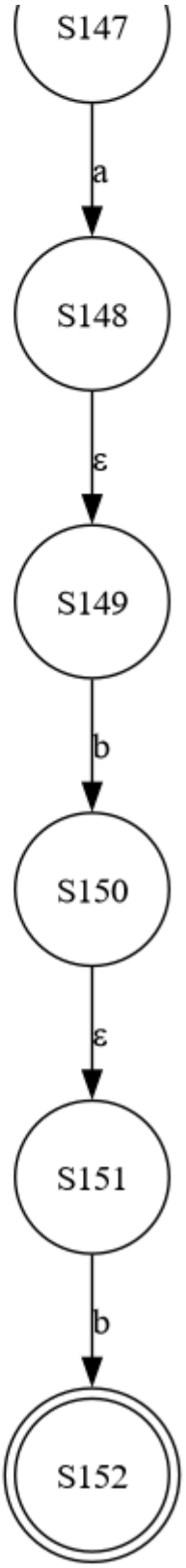
结果

```
Testing pattern: a.b
Matched: ab
Unmatched: abc
Unmatched: abcc
Unmatched: c
Unmatched: abbbbb
Unmatched: abab
Unmatched: bababababaaa

Testing pattern: a|c
Unmatched: ab
Unmatched: abc
Unmatched: abcc
Matched: c
Unmatched: abbbbb
Unmatched: abab
Unmatched: bababababaaa

Testing pattern: a.(b|c)
Matched: ab
Unmatched: abc
Unmatched: abcc
Unmatched: c
Unmatched: abbbbb
Unmatched: abab
Unmatched: bababababaaa

Testing pattern: (a|b)*
Matched: ab
Unmatched: abc
Unmatched: abcc
Unmatched: c
Matched: abbbbb
Matched: abab
Matched: bababababaaa

Testing pattern: (a|b)+
Matched: ab
Unmatched: abc
Unmatched: abcc
Unmatched: c
Matched: abbbbb
Matched: abab
Matched: bababababaaa

Testing pattern: a.b+.c?
Matched: ab
Matched: abc
Unmatched: abcc
Unmatched: c
Matched: abbbbb
Unmatched: abab
Unmatched: bababababaaa
```

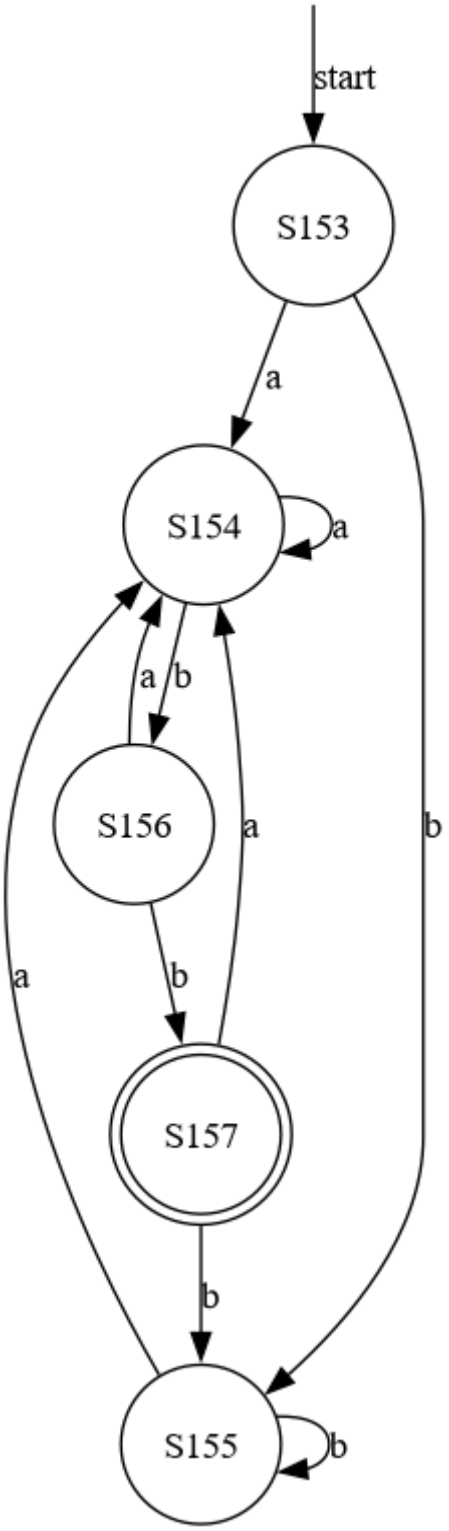
```
Testing pattern: (a.b)*|c+
Matched: ab
Unmatched: abc
Unmatched: abcc
Matched: c
Unmatched: abbbbb
Matched: abab
Unmatched: babababababaaa

Testing pattern: b.(a|b)*.a.a
Unmatched: ab
Unmatched: abc
Unmatched: abcc
Unmatched: c
Unmatched: abbbbb
Unmatched: abab
Matched: babababababaaa

Testing pattern: (a|b)*.a.b.b
Unmatched: ab
Unmatched: abc
Unmatched: abcc
Unmatched: c
Unmatched: abbbbb
Unmatched: abab
Unmatched: babababababaaa
```

NFA:

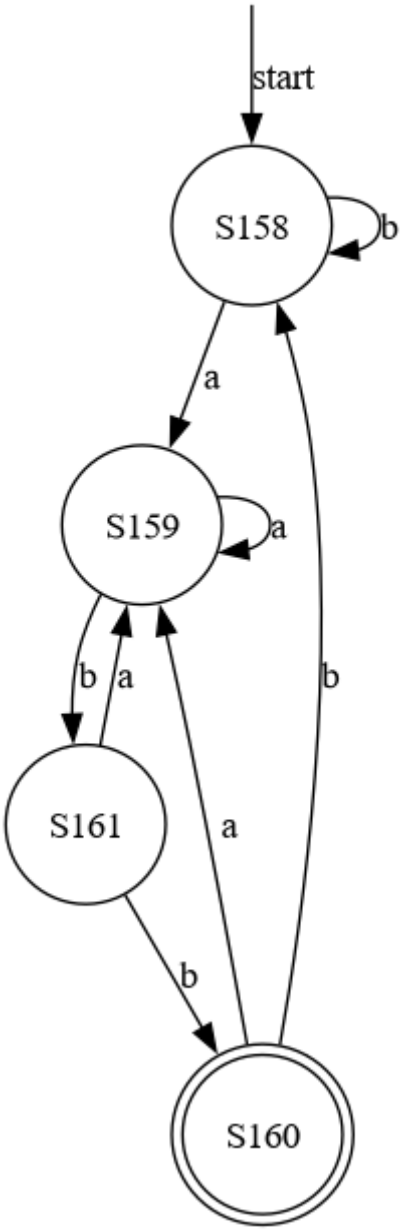

start
S145
ε
S143
ε
ε
S139
S141
a
b
ε
ε
S140
S142
ε
ε
S144
ε
S146
ε



DFA:



最小化 DFA:



所有测试均显示 NFA, DFA, 和最小化 DFA 在所有测试字符串上的结果一致，且与预期匹配结果相符。这表明正规表达式到自动机的转换以及自动机的最小化均实现正确。

实验总结

收获

1. **理论与实践的结合**：通过实际编码和测试，加深了对正规表达式、NFA、DFA 及其最小化理论的理解。
2. **编程技能提升**：在实现算法过程中，提升了 Python 编程能力，特别是在处理复杂数据结构和算法时。
3. **调试能力增强**：在确保转换过程中的正确性和效率方面，增强了代码调试和优化的能力。

挑战

1. **算法的复杂性**：Hopcroft 算法的实现较为复杂，特别是在处理大量状态和符号的情况下。
2. **效率问题**：在初步实现时，面临效率和性能问题，特别是在子集构造和最小化算法中，需要优化算法以处理大规模的输入。
3. **错误处理**：初期测试中发现了几个逻辑错误，特别是在状态转移和最小化算法中，通过反复测试和调整才得以纠正。

总的来说，这个实验不仅提升了对自动机理论的理解，也锻炼了实际的编程和问题解决技能。通过这些实验活动，可以更好地准备应对计算理论或相关领域的更高级课题。